COT 5405 - Analysis of Algorithms

Algorithms Programming Project

Group Members

Sai Sree Ram Ogirala - UFID 87337736

Venkata Satya Sai Subhash Vadlamani - UFID 43268265

Gummi Reddy Jagadeeswara Reddy - UFID 50955833

Fall Semester 2022

Team Members Contribution

1. Sai Sree Ram Ogirala (UFID: 87337736)
   Involved in the Design of Algorithms, Report
2. Venkata Satya Sai Subhash Vadlamani (UFID: 43268265)
   Involved in Design of Algorithm, Coding and Implementation
3. Gummi Reddy Jagadeeswara Reddy (UFID: 50955833)
   Involved in the testing, experimental study

To run our code:

Once we execute the 'make' command to create the 'Stocks' executable file, inorder to run the executable file, we need to use the command 'java Stocks <task_number>'.

Design and Analysis of Algorithm

Algorithm1: Brute force for Problem 1

Design:

Algorithm1 is a brute force algorithm of problem 1 where we try to find the maximum profit out of all the m stocks by considering the maximum profit achieved from each stock and then picking the maximum of all of them. This is brute force since the technique used in finding the maximum profit for each stock is primitive/trivial and elementary where we try all the combinations of buyDay, SellDay for that particular stock and select the pair (buyDay,sellDay) with the maximum profit. This method is the natural approach or the first thought that occurs to the mind for solving this problem. We store the transaction which gives the maximum profit by storing the buyDay, sellDay and in which stock.

Correctness:

Since in this algorithm we consider all the plausible pairs of (buyDay,Sellday) for every stock where sellDay > buyDay and update the maximum profit we achieved so far only if we come across a profit greater than current max profit this solution will always give the correct result for any valid input. The answer returned by the algorithm will return the values of the stock and the transaction taken place for that stock as the answer.

Time complexity: Θ(m∗n^2)

We iterate over every *possible* pair of days for each stock i.e O(n^2) and since there are m stocks the time complexity is in the order of m*n^2.

Space Complexity: Θ(1) //not considering the space needed for input
Except for reading and storing the input values we are not using any additional space for this algorithm. Therefore it takes constant space.

```
Algorithm1( ) {

Read the input for m stocks and n days
max_profit= -infinity
answer={}

for s in 1 to m:
        //find the highest profit for each of the stocks
        for i in 1 to n-1:
                for j in days from i+1 to n:     // check for the price of the stock
on day after the ith day
                 if (stockPrice[s][j] - stockPrice[s][i] > max_profit){
                   max_profit = stockPrice[s][j] - stockPrice[s][i];
                    answer={s,i,j}
                }


return answer
}
```

Algorithm2: Greedy algorithm for problem1

Design:

In the design of this algorithm we try to optimize the time complexity of the existing brute force algorithm we have currently by taking a greedy approach. On observing closely we are able to make an inference that we achieve maximum profit when the difference in price between the sellDay and buyDay is maximum. So it is possible to achieve maximum profit when we greedily aim to consider the minimum price of the stock as we traverse through the prices of a single stock. This can also be considered as the relative minimum, as compared to the prices we have seen so far while traversing through the prices. First we assume the first value of the stock price as the minimum then as we traverse through the stock prices on subsequent days if the stock price is greater then we find the profit and if the profit is greater than the current max_profit which is initially -infinity then we update the max_profit. If the stock price is less than the current minimum stock price I.e buyPrice then we update the current buyDay to this day and the current buyPrice to this price. We repeat this for all stocks.

Correctness:

Since in this algorithm we are constantly in search of the minimum stock price where the buyDay is less than the sellDay we are greedily trying to maximize the profit by choosing the minimum stock price each time. At the end of each row we will arrive at the maximum profit achieved so far. Suppose for a single stock, assume that the maximum profit we got is not the answer. This implies that there is another transaction of buyDay, SellDay for which the profit2 is greater than the maximum profit we got. But , we consider all the scenarios which gives us a profit greater than current profit. Meaning it is not possible to get a profit greater than the profit we achieve at the end of the array. This proves that our assumption is wrong, this implies that profit2=max_profit though it may possible that there is another transaction with same profit (if more than 1 answer is present in the solution)

Time complexity: Θ(m∗n)

We iterate over a stock prices of a single stock by finding out the minimum price to buy the stock and maximum price to sell the stock in a single pass which means it is of the order - O(n). Since there are m stocks the time complexity is in the order of m*n.

Space Complexity: Θ(1)   //not considering the space needed for input
Except for reading and storing the input values we are not using any additional space for this algorithm which is a function of the input values. Therefore it takes constant space.

```
Algorithm2( ) {

Read the input for m stocks and n days
max_profit= -infinity
answer={}



for i in 1 to m{
    currentStock = i ;
    buyPrice = stockPrice[ i ][1];
    buyDay = 1;
    for j in 2 to n{
        if (buyPrice > stockPrice[i][j]){
                buyPrice = stockPrice[i][j];
                buyDay = j ;
        }
        else if (stockPrice[i][j] - buyPrice > max_profit){
            maxProfitStock = currentStock;
            max_profit = stockPrice[i][j]- buyPrice;
            stockBuyDay = buyDay;
            stockSellDay = j;
            answer={maxProfitStock, stockBuyDay, stockSellDay}

        }
    }

}

return answer
}
```

Algorithm3: Dynamic Programming algorithm for solving Problem1

Design:

In this algorithm we divide the problem into overlapping subproblems and arrive at the result by computing the result of the current as a function of the previous already solved problems. This solution is somewhat similar to the previous approach but it differs in the way we arrive at the solution. Here we find the maximum profit to stock prices in the interval of [i,j] using the previous approach and then find the maximum profit in the interval of [i, j+1]. Ultimately, we will have the solution for the interval [1,n]. Assume the maximum profit in the interval [1,1] as 0 then in the next subsequent intervals [1,i] we check if the profit till price[i] is greater than in interval[1,i-1]. If yes, then update this value to the interval[1,i] else we update the interval[1,i] with the value in interval[1,i-1]

Correctness:
The proof of this algorithm can be given with proof of induction. Since this algorithm works for the values of x=[1,n] we try to prove that it gives the correct answer for the value of x=n+1. Since in the algorithm we have given, we try to compute the value of local_maximum[i] as a function of the previous value local_maximum[i-1]. In the same way we can compute the value of local_maximum[i+1] where I=n. This completes the proof.

Time complexity: Θ(m*n)

We can find the maximum profit of a stock in a single pass i.e O(n) and since there are m stocks in total it would be of the order m*n.

Space Complexity: Θ(1)   //not considering the space needed for input
The space complexity of this approach would be O(n) since we are storing the values of the minimum price and maximum profit so far achieved in the arrays of size-n. However, we can do without storing the values of all the previous results, thereby reducing the space complexity to O(1).

Recursive Formulation:
local_maximum[i] is the maximum profit value of a stock in the interval of the first i days
minimum[i] is the value of the minimum value of stock till the ith day

```
{local_maximum[i]}= (for i=1 to n)
                                if i=1 then {Arr[1]}
                                else {max(local_maximum[i-1], Arr[i] - minimum[i-1])}
minimum[i]=         if i=1 then {Arr[1]}
                                else min(minimum[i-1], Arr[i])
```

We can find the local_maximum of a single row in local_maximum[n].

**The below algorithm has Space Complexity O(n) because of the use of minimum[n] and local_maximum[n], but it can be optimized to O(1) as you can see in the next Algorithm which is the optimized algorithm.

```
Algorithm3( ) {

Read the input for m stocks and n days
max_profit= -infinity
answer={}
max_profit=0
for i in 1 to m {

    minimum[n]; local_maximum[n];
    minimum[1]=stockPriceMatrix[i][1]; local_maximum[1]=0;
    stockBuyDay=1;
    stockSellDay=1;
    for j in 2 to n {

        local_maximum[j] = max(local_maximum[ j - 1], stockPriceMatrix[i][j] -
minimum[j - 1]);
        minimum[j] = min(minimum[j - 1], stockPriceMatrix[i][j]);
        if local_maximum[j] > local_maximum[j - 1] {
            stockSellDay = j
        }
        if minimum[j] < minimum[j - 1] {
            stockBuyDay = j
        }
    }
// outside the inner for loop
    if local_maximum[n]>max_profit {
        answer = {i, stockBuyDay, stockSellDay};
        max_profit=local_maximum[n];
    }

}
return answer
}
```

**Instead of storing the values at each step in the array we just update the maximum_profit, min_price.

```
Algorithm3( ) {

Read the input for m stocks and n days
max_profit= -infinity
answer={}
for i in 1 to m{
    currentStock = i ;
    currentStockLowestPrice = stockPriceMatrix[i][0];
    currentStockLowestPriceDay = 1;

    for j in 2 to n{
        sellDay = j ;
        currentProfit = stockPriceMatrix[i][j] - currentStockLowestPrice;
        if (currentProfit > maxProfitSoFar){
            maxProfitSoFar = currentProfit;
            stockBuyDay = currentStockLowestPriceDay;
            stockSellDay = sellDay;
            mostProfitableStock = currentStock;
            answer={mostProfitableStock, stockBuyDay, stockSellDay}
        }
        if (stockPriceMatrix[i][j] < currentStockLowestPrice){
            currentStockLowestPrice = stockPriceMatrix[i][j];
            currentStockLowestPriceDay = sellDay;
        }


    }

}

return answer
}
```

Algorithm4: Brute Force algorithm for solving Problem2

Design:

In this algorithm we recursively call a function to find the minimum price of a stock on a particular day. And similarly, we try to find the maximum price of a stock on all given days greater than the min price found previously. We then use a stack to keep track of the all the transactions made that have resulted in maximum profit.

Correctness:

In this algorithm, since this we are using the brute force approach we are checking for all the possible pairs and then picking the solution with maximum profit. This ensures that we get the correct solution.

Time complexity: Θ(m∗(n^2k))

Space Complexity: Θ(max depth of the recursion) //not considering the space needed for input

Algorithm5: Dynamic programming algorithm for solving Problem2

Design:
In our algorithm first we are considering a 2-D array to store the profits achieved by each stock for each day and for each transaction. We get the maximum profit achieved for the input in the array profit[k][n] where k is the number of transaction and n is the number of days. Once we have the maximum profit we backtrack to extract the solution for the problem to get the transaction used to get the maximum profit.

We use the following recursive formulation:
We update the maximum profit achieved and keep track of the profit achieved by using the formulation below.
Maximum profit so far = {max profit so far ,price[I][j] - price[i][a] + profit[I - 1][a] }

profit[I][j] = max(profit[I][j - 1], maximum profit so far)

Correctness:

Our Algorithm works for k=2,3 transactions in a similar way by extending it to higher orders of k on using proof by induction we can prove similarly that it holds true for I>k. Since we are solving the problem by overlapping sub problems and as we are solving the problem for some k=3 it will work for k>3. Also since we are trying for all possible transactions the algorithm we use is correct and gives the optimal result.

Time complexity: Θ(m∗n^2∗k)
For k transactions we check the valid buyDay and SellDay pairs for a stock, this makes the algorithm in the order of n^2 and since we use another for loop where we iterate for k transactions, this makes it of the order n^2 *k and since we are doing this for m stocks the total time complexity will be m*n^2*k

Space Complexity: Θ(k*n) //not considering the space needed for input
The space complexity of our algorithm is k*n since we are storing the the profit values for a stock for k transactions.

Algorithm6: Dynamic programming algorithm for solving Problem2

Design:

In our algorithm, similar to the previous algorithm we store the profits achieved in a 2 dimensional array, however in this in order to further reduce the time complexity from m∗n^2∗k we make an optimization. Here we basically keep track of difference in price of profits for the m stocks on each day compared to the previous day thus reducing our time complexity. At the end we will have the result stored in the last value of the 2 dimensional array just like in the previous case. And we get the transaction made to get the maximum profit by using a stack and by pushing the buyDay and SellDay to it when we are getting the maximum profit after making a transaction. We

Correctness:

The proof of correctness of this algorithm is quite similar to the previous algorithm. We prove by induction that since our algorithm is working for a value x=k it satisfies and works for x=k+1. The solution given by this algorithm is the optimal solution where we get the transaction with the maximum profit.

Time complexity: $\Theta(m * n * k)$

Since in this algorithm we are checking the profit on each day for each stock at the same time by making a single transaction in a single pass it takes the time which is in the order of m*n and since we are checking for k transactions it will be m*n*k.

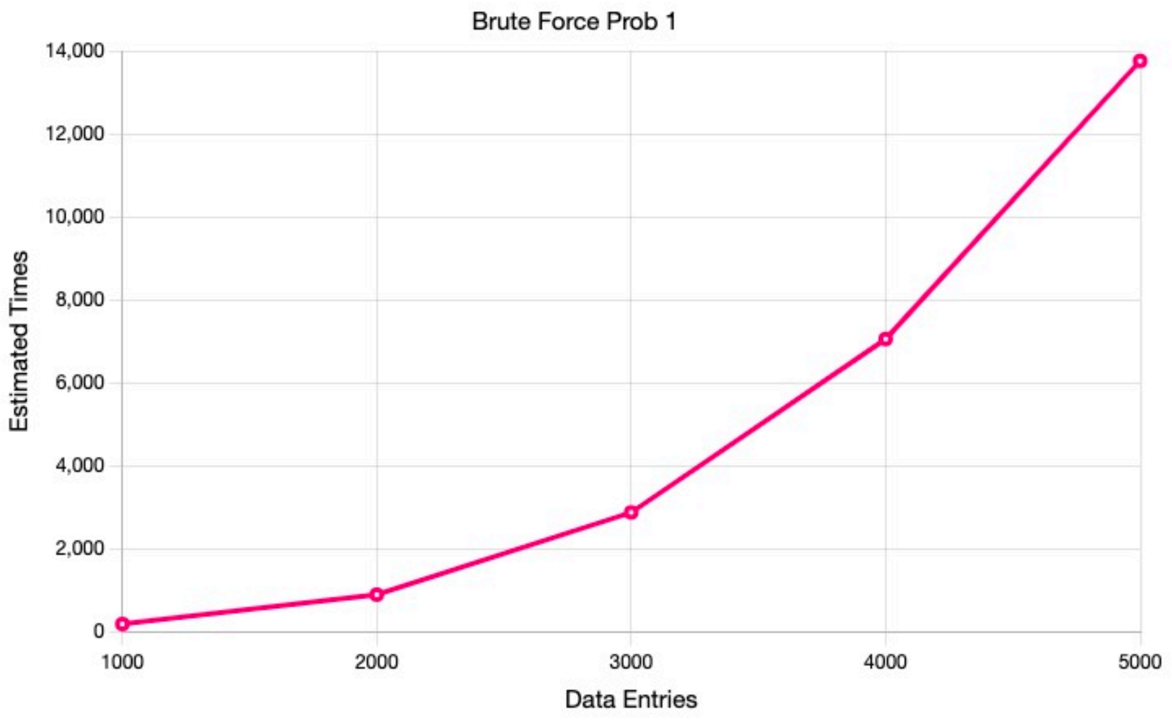
Space Complexity: $\Theta(k*n)$ //not considering the space needed for input
The space complexity of our algorithm is k*n since we are storing the the profit values for a stock for k transactions.

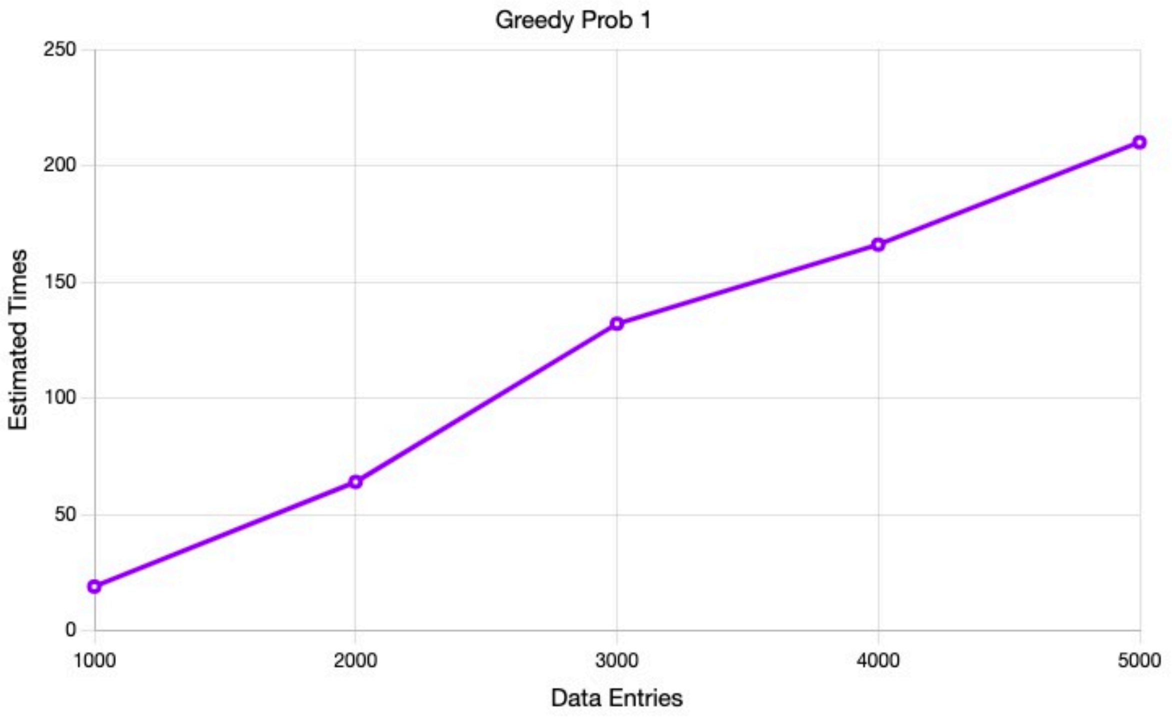
Executable Comparitive Study

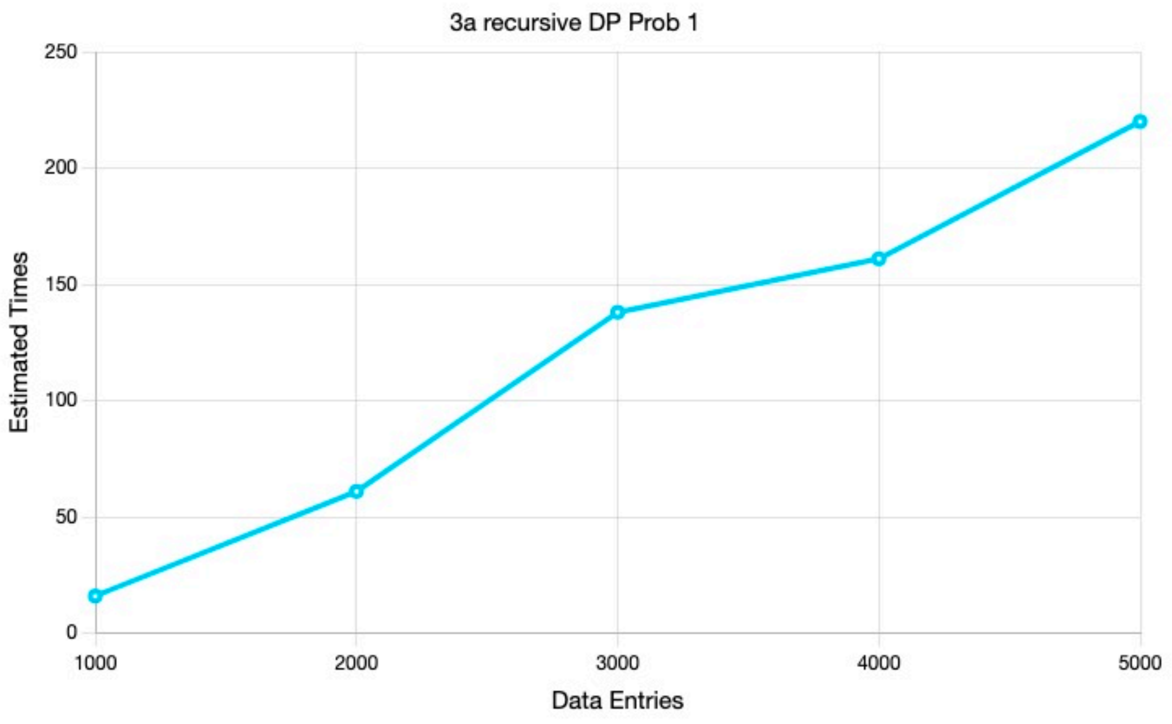
Execution Time in ms

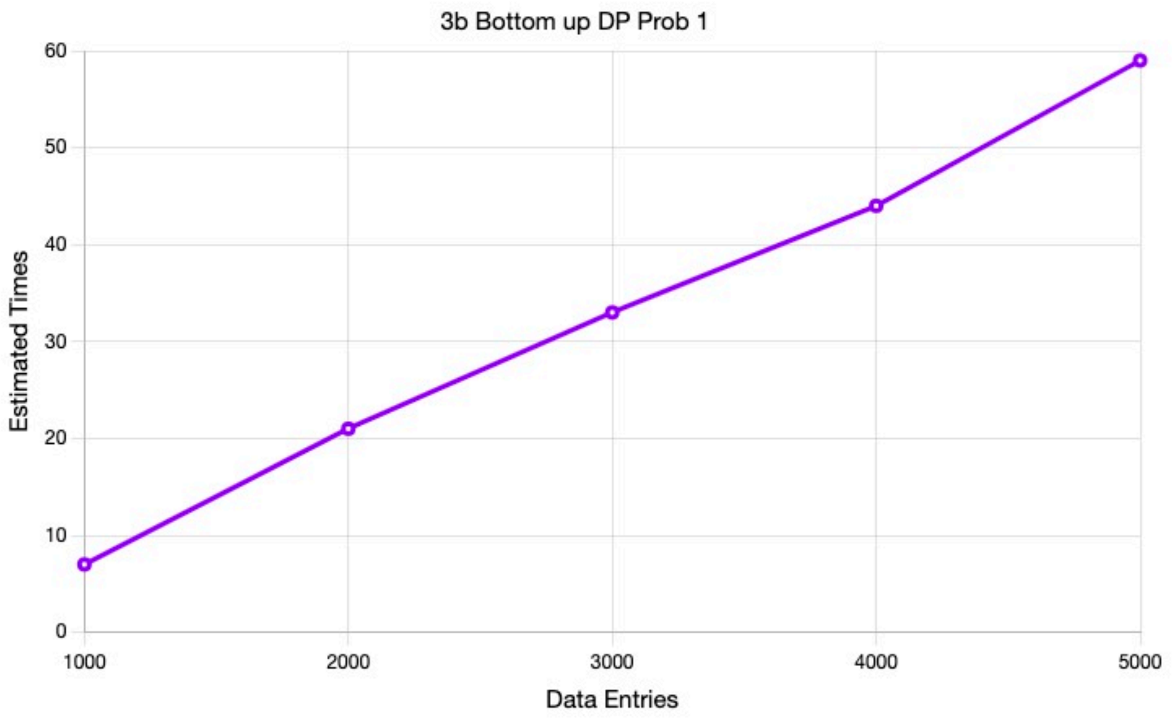
INPUT SIZE	TASK 1	TASK 2	TASK 3A	TASK 3B
1000	198	19	16	7
2000	905	64	61	21
3000	2885	132	138	33
4000	7060	166	161	44
5000	13755	210	220	59

Please omit this space


Brute Force Prob 1
14,000
12,000
10,000
8,000
6,000
4,000
2,000
0
Estimated Times
1000
2000
3000
4000
5000
Data Entries


Greedy Prob 1
250
200
150
100
50
0
Estimated Times
1000
2000
3000
4000
5000
Data Entries


3a recursive DP Prob 1
250
200
150
100
50
0
Estimated Times
1000
2000
3000
4000
5000
Data Entries


3b Bottom up DP Prob 1
60
50
40
30
20
10
0
Estimated Times
1000
2000
3000
4000
5000
Data Entries